DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
UNIVERSITY OF NEBRASKA—LINCOLN

MGG Database-backed Sales System

Computer Science II Project

Eric Le & Brock Melvin

04/22/2021

Version 6.0

This design document covers the information regarding a database-backed sales system that is Objected-Oriented and written in Java for Modern Geek Games to organize their sales records.

Revision History

Version	Description of Change(s)	Author(s)	Date
1.0	Initial draft of this design document	Eric Le & Brock Melvin	2021/02/16
2.0	Improvement & Establishment of classes	Eric Le & Brock Melvin	2021/03/3
3.0	Improvement & Outlining database design	Eric Le & Brock Melvin	2021/03/17
4.0	Improvement & Adding Data Structure	Eric Le & Brock Melvin	2021/03/29
5.0	Improvement & Incorporated ADT List	Eric Le & Brock Melvin	2021/04/12
6.0	Final changes using feedback	Eric Le & Brock Melvin	2021/04/22

Contents

1. Introduction

This database-backed sales system is designed to replace the old collection of Excel sheets and macros used previously by Modern Geek Games (MGG) in order to modernize and compete in the digital-copy only market. The system will identify items under three main categories, products, services, and subscriptions, with a unique alphanumeric code and a name.

1.1 Purpose of this Document

The purpose of this document is to outline the design of a Sales system and provide a run-through of the structure and its applications.
The main intention is to sell items under three categories with organization -

- **Products** - games physical and digital with a base price, a used price, and gift card prices
- **Services** - performed by MGG staff; the cost is hourly rate multiplied by the number of hours
- **Subscriptions** - online game service, memberships, or third-party services. Which are paid in annuals fees

1.2 Scope of the Project

The database-backed sales system made for Minerva Campbell will allow Modern Geek Games (MGG) to have the ability to manage sales at their stores utilizing the MGG's business rules where items such as products, services, and subscriptions are taken in account based on their specificity towards their final price. Additionally, the customer's membership status will be accounted for as well towards the overall sales price. This system will be incorporated with the use of Java and SQL for designing the classes and database that are necessary for the functionality of the database-backed sales system.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

Electronic Data Interchange - The computer-to-computer exchange of business documents in a standard electronic format between business partners

Application Programming Interface - A software intermediary that allows two applications to talk to each other

HashMap - A class in the java.util package that provides the basic implementation of the Map interface of Java where it stores data in (Key, Value) pairs, and can be accessed by an index of another type, say an integer.

1.3.2 Abbreviations & Acronyms

MGG - Modern Geek Games
XML - Extensible Markup Language
JSON - JavaScript Object Notation
EDI - Electronic Data Interchange
ADT - Abstract Data Type
API - Application Programming Interface

2. Overall Design Description

This document is made in multiple phases during the project -

Phase I: Data Representation & Electronic Data Interchange (EDI) – In this first phase, entities are created and incorporated for the MGG Database-backed Sales System as a starting foundation. Parsers are designed to process data from inputted flat files and in turn, output a readable XML and JSON format.

Phase II: Summary Report – Entities are developed even further with the use of inheritance to provide a sense of hierarchy to the database-backed sales system. This allows a summary report of salespersons, stores, and each individual item of a sale to be generated.

Phase III: Database Design – Rather than relying on a flat file for information, a database design is created to allow more malleable forms of information in regards to a sale. Entities are now modeled in the database in a way where they have an actual relation with each other to support the MGG's business model.

Phase IV: Database Connectivity – Instead of parsing data from general flat files, the database-backed sales system will be altered to parse data from the more effective database design created in Phase III.

Phase V: Database Persistence – An API will be implemented and used to persist or save data to the database

Phase VI: Sorted List ADT – A sorted list ADT will be designed and implemented. Additionally, it will be integrated into the application.

The classes consist of entities such as people, items, and sales which is further detailed in 3.1 Database Design in regards to their categories and particular attributes. Additionally, there are writer and reader classes designed and implemented within the MGG's Database-backed Sales System. The reader classes contain a general parser for flat files and a database parser that reads directly from the MGG's database design. The writer classes such as DataConverter and SalesReport serve the purpose of exporting information into a readable format. DataConverter outputs data into XML and JSON formats, and SalesReport generates a summary report of salespersons, stores, and each individual item of a sale.

The database design of having an efficient method of processing and outputting the necessary information needed for each sale in a way that is organized and withholds all of the necessary applications of taxes and discounts depending on the person and item allows the program to abide by all of the business rules belonging to MGG. The database program will improve MGG's ability to compete in the digital market as well as to create a more efficient system for how MGG will function in regards to their sales due to organisability from the system.

2.1 Alternative Design Options

A HashMap design was considered for the association of a manager with a particular store, but was ultimately discarded as a result of a HashMap being unnecessary due to person, item, and store codes being provided within .csv files that connect such instances. While a HashMap would be a more advanced design for multiple situations, the database design only calls for an alphanumeric code in a connection that has already been

established or in regards to a sale. Searching simply for a particular code stated within the .csv files is more efficient and easier without needing unnecessary effort.

An Address class was utilized within the code, because the Address class was able to reduce the amount of code needed for arbitrary information of every class such as person and store. Without an Address class, there would be multiple lines of code just for simple strings like street and country when a single class can easily handle such repeated information.

A table per class/subclass strategy was initially thought of, but in the end, a "single table inheritance strategy," was decided upon because a table per class/subclass strategy would overcomplicate the database design, and a "single table inheritance strategy" would allow a more efficient approach with types already being established that can be discriminated.

After incorporating a query within mySQL to detect multiple instances of an item within a sale, such as a purchase for a game appearing twice without the quantity factor changing, there was a realization that this query shouldn't even be required in the first place. Therefore, a unique pair constraint between a Sale and an Item was incorporated in SaleItem to prevent such instances.

3. Detailed Component Description

The program will organize the three main categories: Person, Store, and Items, and create subcategories to better perform specific functions.

3.1 Database Design

The database will have three main entities: Person, Store, and Items. Person will have the subcategories of Employee, Platinum, Gold, and a regular Customer, each with their own unique characteristics towards their overall payment and role. An Employee has a 15% discount, Platinum Customer has a 10% discount, Gold Customer has a 5% discount, and a regular Customer possesses no discount. Also, only individuals with an Employee designation type can fulfill the role as a salesperson while the other types cannot. Additionally, Items will be organized under three subcategories of Products (split further into used products, new products, and gift cards), Services, and Subscriptions. These Item subcategories will have their own effects on the tax that is charged to the customer. For new products, used products, and gift cards, they will all be taxed 7.25%, but in regards to used products, they will be 80% of their base price, and gift cards will have their price established by the customer at a sale transaction. Services receive a 2.85% tax and retain information concerning the employee conducting the service, the amount of hours, and the hourly charge. Finally, subscriptions do not possess a tax rate and will withhold the time period of the subscriptions start date and end date for the sale to determine its cost.

The Entity-Relation diagram for the database demonstrates hierarchy in regards to the tables used in relation to a sale along with primary key ids and foreign key ids to connect tables such as Sale with SaleItem or Sale with Store to better organize and connect sale data collected from MGG’s business.

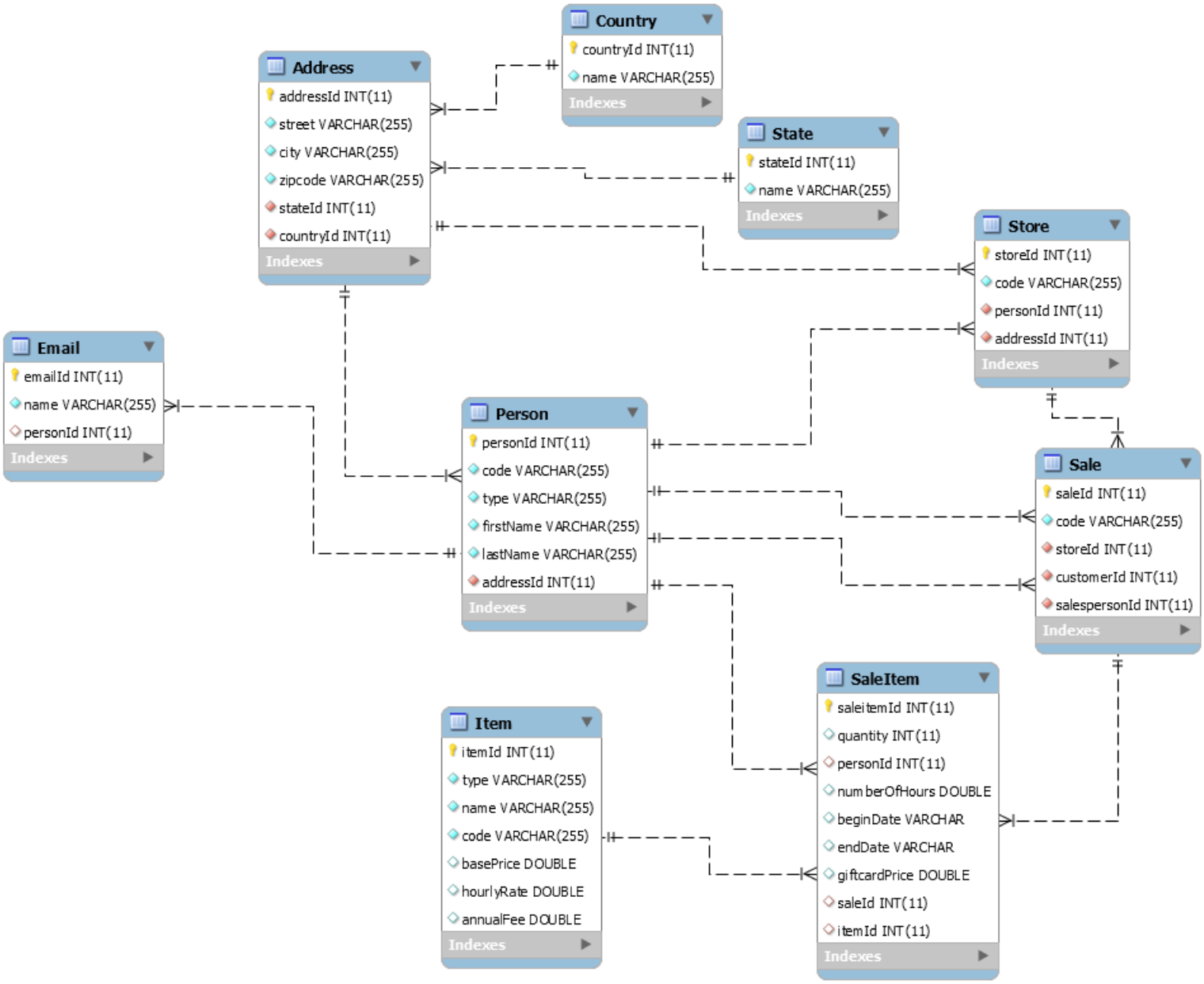

Figure 1: Entity-Relation Diagram of the database-design schema

3.1.1 Component Testing Strategy

To ensure that the inputted information provided within the .csv files are placed in their rightful categories, the program's XML and JSON outputs must match the correct tags for a Person or Item that is presented. The non-trivial test cases created should match exactly with the XML and JSON outputs created by the program, including correct format. The attributes belonging to each category concerning MGG's business rules will be further expanded upon in testing in section "3.2.1 Component Testing Strategy."

To verify the functionality of our database design, non-trivial test data is inserted into the tables (Sale, Person, Store, Item, etc.), and if the inputted information is retained along with their unique attributes as determined by the single table inheritance strategy, then our database design is sufficient. Queries such as those that are used to select certain information from the database to pull from were utilized to verify the database's correct design.

An incorporation of a query to detect multiple instances of an item within a sale caused a realization that a unique constraint was needed in order to prevent such multiple instances from occuring in the first place. So, after implementing the unique constraint within SaleItem regarding Sale and Item, a chance for multiple instances was completely prevented.

The sales report outputted by the system using the database parser matched the expected output using test data. Additionally, the output matched with the output produced by the flat file reader parser originally created in phase II. Therefore, this verified that the database had a good design and functioned correctly.

3.2 Class/Entity Model

The UML Diagram below demonstrates the hierarchy present within the database program for MGG regarding its entities like people and items as well as their relationships towards function or price impaction which is highlighted more in detail within section 3.1.

The MGG Database-backed Sales System possesses two main drivers which is SalesReport and DataConverter. SalesReport produces a summary report of salespersons, stores, and detailed reports of individual items belonging to a sale. DataConverter, ultimately, creates a readable outputted XML and JSON formatted data. The program essentially has two main categories that support the functionality of these main drivers which are readers and writers. The readers consist of Parser, DatabaseInfo, and DatabaseParser which have the purpose of taking in data. The Parser is utilized to read inputted data via .csv files while the DatabaseParser takes information straight from the database utilizing the user's credentials provided in DatabaseInfo. The writer's consist of JSONConverter, XMLConverter, and SaleGenerator. The writers are meant for the purpose of formatting and outputting information to the user in a readable format without having to learn the inner workings of the program. The SaleGenerator class outputs the summary report of a salesperson and stores, and does grand calculations which cannot be done in the individual classes where each salesperson or store holds the sales respective to themselves.

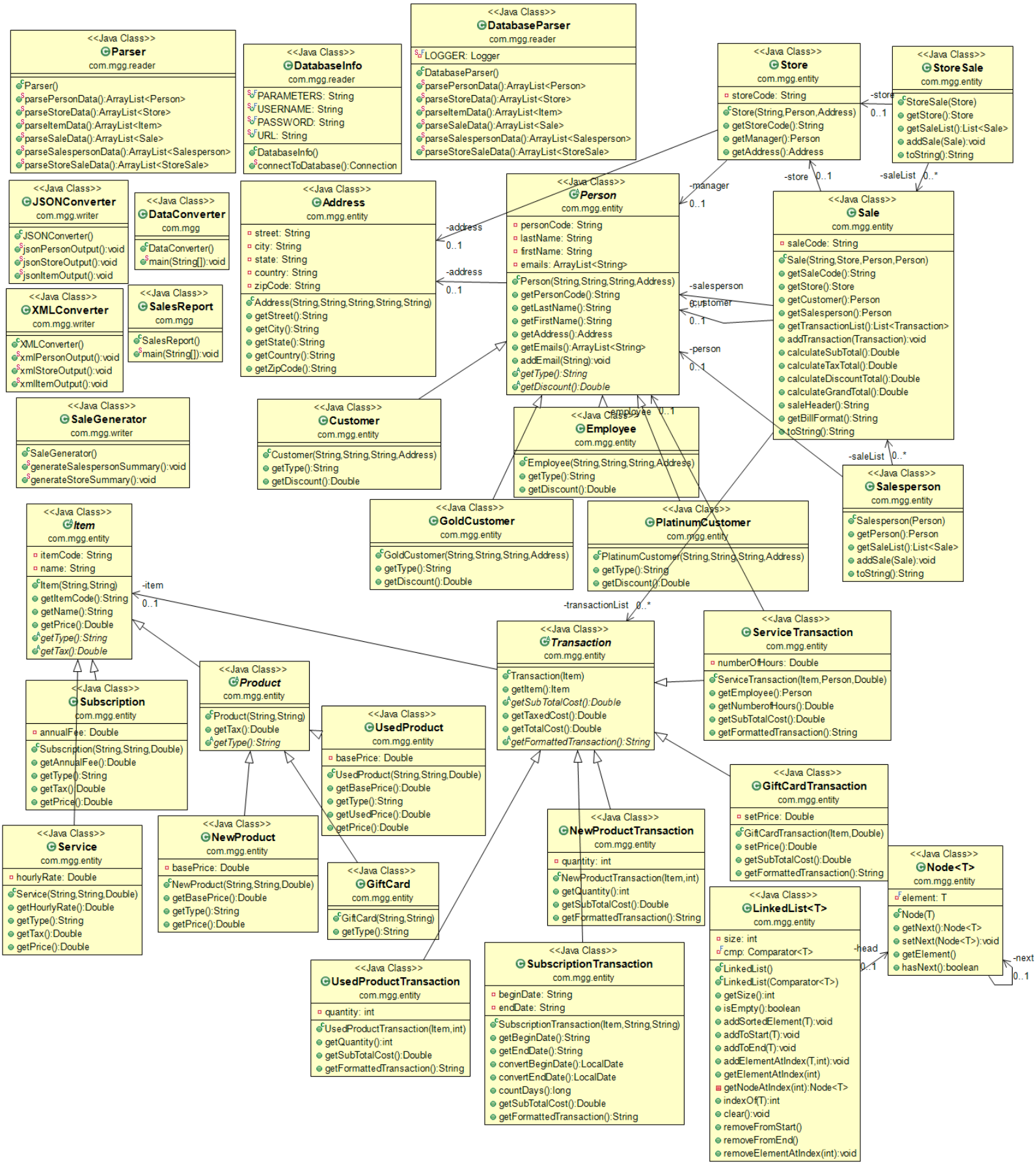
<<Java Class>>
Parser
com.mgg.reader
Parser()
parsePersonData():ArrayList<Person>
parseStoreData():ArrayList<Store>
parseItemData():ArrayList<Item>
parseSaleData():ArrayList<Sale>
parseSalespersonData():ArrayList<Salesperson>
parseStoreSaleData():ArrayList<StoreSale>
<<Java Class>>
DatabaseInfo
com.mgg.reader
PARAMETERS: String
USERNAME: String
PASSWORD: String
URL: String
DatabaseInfo()
connectToDatabase():Connection
<<Java Class>>
DatabaseParser
com.mgg.reader
LOGGER: Logger
DatabaseParser()
parsePersonData():ArrayList<Person>
parseStoreData():ArrayList<Store>
parseItemData():ArrayList<Item>
parseSaleData():ArrayList<Sale>
parseSalespersonData():ArrayList<Salesperson>
parseStoreSaleData():ArrayList<StoreSale>
<<Java Class>>
Store
com.mgg.entity
storeCode: String
Store(String,Person,Address)
getStoreCode():String
getManager():Person
getAddress():Address
-store
0..1
<<Java Class>>
StoreSale
com.mgg.entity
StoreSale(Store)
getStore():Store
getSaleList():List<Sale>
addSale(Sale):void
toString():String
<<Java Class>>
JSONConverter
com.mgg.writer
JSONConverter()
jsonPersonOutput():void
jsonStoreOutput():void
jsonItemOutput():void
<<Java Class>>
DataConverter
com.mgg
DataConverter()
main(String[]):void
<<Java Class>>
Address
com.mgg.entity
street: String
city: String
state: String
country: String
zipCode: String
Address(String,String,String,String,String)
getStreet():String
getCity():String
getState():String
getCountry():String
getZipCode():String
-address
0..1
-address
0..1
<<Java Class>>
Person
com.mgg.entity
personCode: String
lastName: String
firstName: String
emails: ArrayList<String>
Person(String,String,String,Address)
getPersonCode():String
getLastName():String
getFirstName():String
getAddress():Address
getEmails():ArrayList<String>
addEmail(String):void
getType():String
getDiscount():Double
-manager
0..1
-store
0..1
-saleList
0..*
<<Java Class>>
Sale
com.mgg.entity
saleCode: String
Sale(String,Store,Person,Person)
getSaleCode():String
getStore():Store
getCustomer():Person
getSalesperson():Person
getTransactionList():List<Transaction>
addTransaction(Transaction):void
calculateSubTotal():Double
calculateTaxTotal():Double
calculateDiscountTotal():Double
calculateGrandTotal():Double
saleHeader():String
getBillFormat():String
toString():String
-salesperson
0..1
-customer
0..1
-person
0..1
<<Java Class>>
XMLConverter
com.mgg.writer
XMLConverter()
xmlPersonOutput():void
xmlStoreOutput():void
xmlItemOutput():void
<<Java Class>>
SalesReport
com.mgg
SalesReport()
main(String[]):void
<<Java Class>>
SaleGenerator
com.mgg.writer
SaleGenerator()
generateSalespersonSummary():void
generateStoreSummary():void
<<Java Class>>
Customer
com.mgg.entity
Customer(String,String,String,Address)
getType():String
getDiscount():Double
<<Java Class>>
Employee
com.mgg.entity
Employee(String,String,String,Address)
getType():String
getDiscount():Double
employee
0..1
-saleList
0..*
<<Java Class>>
Salesperson
com.mgg.entity
Salesperson(Person)
getPerson():Person
getSaleList():List<Sale>
addSale(Sale):void
toString():String
<<Java Class>>
GoldCustomer
com.mgg.entity
GoldCustomer(String,String,String,Address)
getType():String
getDiscount():Double
<<Java Class>>
PlatinumCustomer
com.mgg.entity
PlatinumCustomer(String,String,String,Address)
getType():String
getDiscount():Double
<<Java Class>>
Item
com.mgg.entity
itemCode: String
name: String
Item(String,String)
getItemCode():String
getName():String
getPrice():Double
getType():String
getTax():Double
-item
0..1
-transactionList
0..*
<<Java Class>>
Transaction
com.mgg.entity
Transaction(Item)
getItem():Item
getSubTotalCost():Double
getTaxedCost():Double
getTotalCost():Double
getFormattedTransaction():String
<<Java Class>>
ServiceTransaction
com.mgg.entity
numberOfHours: Double
ServiceTransaction(Item,Person,Double)
getEmployee():Person
getNumberofHours():Double
getSubTotalCost():Double
getFormattedTransaction():String
<<Java Class>>
Subscription
com.mgg.entity
annualFee: Double
Subscription(String,String,Double)
getAnnualFee():Double
getType():String
getTax():Double
getPrice():Double
<<Java Class>>
Product
com.mgg.entity
Product(String,String)
getTax():Double
getType():String
<<Java Class>>
UsedProduct
com.mgg.entity
basePrice: Double
UsedProduct(String,String,Double)
getBasePrice():Double
getType():String
getUsedPrice():Double
getPrice():Double
<<Java Class>>
GiftCardTransaction
com.mgg.entity
setPrice: Double
GiftCardTransaction(Item,Double)
setPrice():Double
getSubTotalCost():Double
getFormattedTransaction():String
<<Java Class>>
Service
com.mgg.entity
hourlyRate: Double
Service(String,String,Double)
getHourlyRate():Double
getType():String
getTax():Double
getPrice():Double
<<Java Class>>
NewProduct
com.mgg.entity
basePrice: Double
NewProduct(String,String,Double)
getBasePrice():Double
getType():String
getPrice():Double
<<Java Class>>
GiftCard
com.mgg.entity
GiftCard(String,String)
getType():String
<<Java Class>>
NewProductTransaction
com.mgg.entity
quantity: int
NewProductTransaction(Item,int)
getQuantity():int
getSubTotalCost():Double
getFormattedTransaction():String
<<Java Class>>
Node<T>
com.mgg.entity
element: T
Node(T)
getNext():Node<T>
setNext(Node<T>):void
getElement()
hasNext():boolean
-next
0..1
<<Java Class>>
LinkedList<T>
com.mgg.entity
size: int
cmp: Comparator<T>
LinkedList()
LinkedList(Comparator<T>)
getSize():int
isEmpty():boolean
addSortedElement(T):void
addToStart(T):void
addToEnd(T):void
addElementAtIndex(T,int):void
getElementAtIndex(int)
getNodeAtIndex(int):Node<T>
indexOf(T):int
clear():void
removeFromStart()
removeFromEnd()
removeElementAtIndex(int):void
-head
0..1
<<Java Class>>
UsedProductTransaction
com.mgg.entity
quantity: int
UsedProductTransaction(Item,int)
getQuantity():int
getSubTotalCost():Double
getFormattedTransaction():String
<<Java Class>>
SubscriptionTransaction
com.mgg.entity
beginDate: String
endDate: String
SubscriptionTransaction(Item,String,String)
getBeginDate():String
getEndDate():String
convertBeginDate():LocalDate
convertEndDate():LocalDate
countDays():long
getSubTotalCost():Double
getFormattedTransaction():String

Figure 2: UML Diagram of the database program hierarchy.

3.2.1 Component Testing Strategy

The entities or classes within the database program will be tested through whether a Person or Item is appropriately designated for a particular sale along with their attributes. A customer of a particular status will have their own particular benefits, and the same could be said for an item which possesses its own rules for taxing and price determination. The grand total created by the non-trivial test cases that account for MGG's business rules of taxes and discounts should match the output produced by the program in the same format.

3.3 Database Interface

The MGG's Database-backed Sales System will load data straight from the database by using the Java Database Connectivity API (JDBC) with mySQL Connector. Through this, the entities such as people and items within the program will be established via the information gained through the API loading the database. Null information is dealt through ignoring certain information depending on the type discriminator depicted within the database. If somehow errors such as fraud, unknown, or missing information are encountered during the parsing process, log4j, a logging system, is utilized to document the error. The SalesReport driver class will still retain its original functionality of generating the same summary and detail report as produced in Phase 2, though instead of reading data from flat files, the information is gained directly from the database instead which allows for more malleable data with set relations to one another, improving MGG's work ethic concerning sale management.

The database that was designed and implemented in Phase 3 will interact with the application created in Phases 1 and 2. Thus the application can be used to create, retrieve, update, and delete objects stored in the database through incorporating defined methods. Now that the API is more general, it can more easily be implemented into different applications to interact with a database. Users of the API are not expected to know the inner workings of the database or of the API to be able to properly use the system.

3.3.1 Component Testing Strategy

All of the data that would have been parsed from the flat files in Phase 2 will match the same data parsed via the database design created in Phase 3. This will present that the information is retained even though the method of reading or parsing data has altered, ultimately approving the program's ability to connect to the database and its design.

The integration of the database design will be tested through if the API persists data through the database. Information changed through the API should be present within our database. How the data is manipulated and where the data is added have to exactly match with the user's intentions in order to follow MGG's business system.

3.4 Design & Integration of Data Structures

To keep track of sales, the system implements and incorporates a sorted list ADT. This ADT will be implemented using a linked list that is generalized to hold other objects as well. This is done by parameterizing the ADT. The linked list implementation will keep track of a head sale and the next sale. Every other sale will have a link to the next one, except for the last one. It will also keep track of the total number of sales in the list at any given time. When a sale is added or removed from the list, the links to the other sales

are either added or removed, making the list the correct size and still maintaining proper functionality. The list of sales can be ordered alphabetically by customer name, by total value of the sale, and by store.

3.4.1 Component Testing Strategy

The linked list ADT will be created and tested using test data. The linked list should be able to store a head sale, and each sale in the list should have a reference to the next sale. The last sale should not have a reference to another sale. The list should be able to be updated (sales added or removed) and still function correctly, showing the correct number of sales in the correct order. A summary table will be printed for each of the three orderings of sales to ensure correctness.

An expected output using trivial data was created which consisted of three lists: a sales summary sorted by customer, a sales summary sorted by total, and a sales summary sorted by store then by sales person. By adding elements of an array into the created linked list for MGG into an initial sorted position, an output was generated by the program which matched exactly with the expected output using test data.

3.5 Changes & Refactoring

In the initial phase, the Items and Persons class were generalized. Then, in phase two, with the use of inheritance, they were narrowed down into more specific classes to allow special functions to be performed unique to a particular person or item such as products and services or employee and platinum customer which could not have been done with a generalized class.

In Phase 2, Sales and Persons possessed a leaky abstraction where a list was required for a constructor for the two classes, but was later fixed in Phase 3 with an add method called addEmail and addTransaction which prevented the need to overcomplicate and provide information of a list.

In Phase 3, a Salesperson class and a StoreSale class were established to allow an employee and store to withhold a list of sales that belong to a specific entity for a better design made in Phase 2. This gives the user the ability to discern which sales belong to which instance without much understanding of the inner workings of the program. In other words, it allows for efficiency.

In Phase 4, foreign key constraints for storeId, customerId, and salespersonId within the Sales table were added to ensure a solid connection between the Store, Customer, and Salesperson tables. Additionally, the application of the Date object present in MySQL was ultimately replaced with a String object due to the fact that the java code had already implemented a method that converted a provided string into a date, therefore it would have been less efficient and unnecessary to go out of the way and utilize MySQL's date object.

4. Bibliography

[1] *Xstream.* (2021). Retrieved February 23, 2021, from http://x-stream.github.io/

[2] *Gson.* (2008). Retrieved February 23, 2021, from https://github.com/google/gson

[3] MySQL *Connector/J.* (2021). Retrieved April 5, 2021, from https://dev.mysql.com/downloads/

[4] *Apache Log4j 2* (2021). Retrieved April 5, 2021 from https://logging.apache.org/log4j/2.x/